Array-OL Revisited, Multidimensional Intensive Signal Processing Specification

Pierre Boulet

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Array-OL Revisited, Multidimensional Intensive Signal Processing Specification

Pierre Boulet

N° 6113 — version 2

version initiale Janvier 2007 — version révisée Février 2007

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6113--FR+ENG

Array-OL Revisited, Multidimensional Intensive Signal Processing Specification

Pierre Boulet*

Thème COM — Systèmes communicants
Projet DaRT

Rapport de recherche n° 6113 — version 2 — version initiale Janvier 2007 — version révisée Février 2007 — 24 pages

Abstract: This paper presents the Array-OL specification language. It is a high-level visual language dedicated to multidimensional intensive signal processing applications. It allows to specify both the task parallelism and the data parallelism of these applications on focusing on their complex multidimensional data access patterns. This presentation includes several extensions and tools developed around Array-OL during the last few years and discusses the mapping of an Array-OL specification onto a distributed heterogeneous hardware architecture.

Key-words: Array-OL, parallelism, data parallelism, multidimensional signal processing, mapping

Warning: the figures of this revised version use transparency. They are much prettier and readable than those of the initial version but they may cause trouble when printed or viewed with old software. This version also includes hyperlinks.

* Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, Cité Scientifique, 59655 Villeneuve d'Ascq, France

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 ORSAY Cedex (France)
Téléphone : +33 1 72 92 59 00 — Télécopie : +33 1 60 19 66 08

Array-OL revisité, spécification de traitements de signal multidimensionnel

Résumé : Cet article présente le langage de spécification Array-OL. C'est un langage visuel de haut niveau dédié aux applications de traitement de signal intensif. Il permet de spécifier à la fois le parallélisme de tâches et le parallélisme de données de ces applications avec un focus particulier sur les motifs complexes d'accès aux données multidimensionnelles. Cette présentation inclut plusieurs extensions et outils développés autour d'Array-OL ces dernières années et étudie le problème du placement d'une spécification Array-OL sur une architecture matérielle distribuée et hétérogène.

Mots-clés : Array-OL, parallélisme, parallélisme de données, traitement de signal multidimensionnel, placement

1 Introduction

Computation intensive multidimensional applications are predominant in several application domains such as image and video processing or detection systems (radar, sonar). In general, intensive signal processing applications are multidimensional. By multidimensional, we mean that they primarily manipulate multidimensional data structures such as arrays. For example, a video is a 3D object with two spatial dimensions and one temporal dimension. In a sonar application, one dimension is the temporal sampling of the echoes, another is the enumeration of the hydrophones and others such as frequency dimensions can appear during the computation. Actually, such an application manipulates a stream of 3D arrays.

Dealing with such applications presents a number of difficulties:

- Very few models of computation are multidimensional.
- The patterns of access to the data arrays are diverse and complex.
- Scheduling these applications with bounded resources and time is challenging, especially in a distributed context.

When dealing with parallel heterogeneous and constrained platforms and applications, as it is the case of embedded systems, the use of a formal model of computation (MoC) is very useful. Edwards et al. [11] and more recently Jantsch and Sander [13] have reviewed the MoCs used for embedded system design. These reviews classify the MoCs with respect to the time abstraction they use, their support for concurrency and communication modeling. In our application domain there is little need for modeling state as the computations are systematic, the model should be data flow oriented. On the contrary, modeling parallelism, both task and data parallelism, is mandatory to build efficient implementations. More than a concrete representation of time, we need a way to express precedence relations between tasks. We focus on a high level of abstraction where the multidimensional data access patterns can be expressed. We do not look for a programming language but for a specification language allowing to deal with the multidimensional arrays easily. The specification has to be deadlock free and deterministic by construction, meaning that all feasible schedules compute the same result. In their review of models for parallel computation [26] Skillicorn and Talia classify the models with respect to their abstraction level. We aim for the second most abstract category which describes the full potential parallelism of the specification (the most abstract category does not even express parallelism). We want to stay at a level that is completely independent on the execution platform to allow reuse of the specification and maximal search space for a good schedule.

As far as we know, only two MoCs have attempted to propose formalisms to model and schedule such multidimensional signal processing applications: MDSDF (MultiDimensional Synchronous Dataflow) [4, 21, 24, 25] and Array-OL (Array Oriented Language) [6, 7] . MDSDF and its follow-up GMDSDF (Generalized MDSDF) have been proposed by Lee and Murthy. They are extensions of the SDF model proposed by Lee and Messerschmitt [19, 20]. Array-OL has been introduced by Thomson Marconi Sonar and its compilation has been studied by Demeure, Soula, Dumont et al. [1, 7, 8, 27, 28]. Array-OL is a specification language allowing to express all the

parallelism of a multidimensional application, including the data parallelism, in order to allow an efficient distributed scheduling of this application on a parallel architecture. The goals of these two propositions are similar and although they are very different on their form, they share a number of principles such as:

- Data structures should make the multiple dimensions visible.
- Static scheduling should be possible with bounded resources.
- The application domain is the same: intensive multidimensional signal processing applications.

A detailed comparison of these two models is available in [9].

An other language worth mentioning is Alpha, proposed by Mauras [23], a functional language based on systems of recurrent equations [16]. Alpha is based on the polyhedral model, which is extensively used for automatic parallelization and the generation of systolic arrays. Alpha shares some principles with Array-OL:

- Data structures are multidimensional: union of convex polyhedra for Alpha and arrays for Array-OL.
- Both languages are functional and single assignment.

With respect to the application domain, arrays are sufficient and more easily handled by the user than polyhedra. Some data access patterns such as cyclic accesses are more easily expressible in Array-OL than in Alpha. And finally, Array-OL does not manipulate the indices directly. In the one hand that restricts the application domain but in the other hand that makes it more abstract and more focused on the main difficulty of intensive signal processing applications: data access patterns.

The purpose of this paper is to present in the most comprehensive and pedagogical way the Array-OL model of specification. Departing from the original description of Array-OL (only available in French), we present an integrated view of the language including the various extensions that were made over the years and a more “modern” vocabulary. Section 2 will define the core language. Its projection to an execution model will be discussed in section 3 and we will present a number of extensions of Array-OL in section 4.

2 Core language

As a preliminary remark, Array-OL is only a specification language, no rules are specified for executing an application described with Array-OL, but a scheduling can be easily computed using this description.

2.1 Principles

The initial goal of Array-OL is to give a mixed graphical-textual language to express multidimensional intensive signal processing applications. As said before, these applications work on multidimensional arrays. The complexity of these applications does not come from the elementary functions they combine, but from their combination by the way they access the intermediate arrays. Indeed, most of the elementary functions are sums, dot products or Fourier transforms, which are well known and often available as library functions. The difficulty and the variety of these intensive signal processing applications come from the way these elementary functions access their input and output data as parts of multidimensional arrays. The complex access patterns lead to difficulties to schedule these applications efficiently on parallel and distributed execution platforms. As these applications handle huge amounts of data under tight real-time constraints, the efficient use of the potential parallelism of the application on parallel hardware is mandatory.

From these requirements, we can state the basic principles that underly the language:

- All the potential parallelism in the application has to be available in the specification, both *task parallelism* and *data parallelism*.
- Array-OL is a *data dependence expression* language. Only the true data dependences are expressed in order to express the full parallelism of the application, defining the minimal order of the tasks. Thus any schedule respecting these dependences will lead to the same result. The language is deterministic.
- It is a *single assignment* formalism. No data element is ever written twice. It can be read several times, though. Array-OL can be considered as a first order functional language.
- Data accesses are done through sub-arrays, called *patterns*.
- The language is *hierarchical* to allow descriptions at different granularity levels and to handle the complexity of the applications. The data dependences expressed at a level (between arrays) are approximations of the precise dependences of the sub-levels (between patterns).
- The spatial and temporal dimensions are treated equally in the arrays. In particular, time is expanded as a dimension (or several) of the arrays. This is a consequence of single assignment.
- The arrays are seen as tori. Indeed, some spatial dimensions may represent some physical tori (think about some hydrophones around a submarine) and some frequency domains obtained by FFTs are toroidal.

The semantics of Array-OL is that of a first order functional language manipulating multidimensional arrays. It is not a data flow language but can be projected on such a language.

As a simplifying hypothesis, the application domain of Array-OL is restricted. No complex control is expressible and the control is independent of the value of the data. This is realistic in the given application domain, which is mainly data flow. Some efforts to couple control flows and data flows expressed in Array-OL have been done in [18] but are outside the scope of this paper.

The usual model for dependence based algorithm description is the dependence graph where nodes represent tasks and edges dependences. Various flavors of these graphs have been defined. The expanded dependence graphs represent the task parallelism available in the application. In order to represent complex applications, a common extension of these graph is the hierarchy. A node can itself be a graph. Array-OL builds upon such hierarchical dependence graphs and adds repetition nodes to represent the data-parallelism of the application.

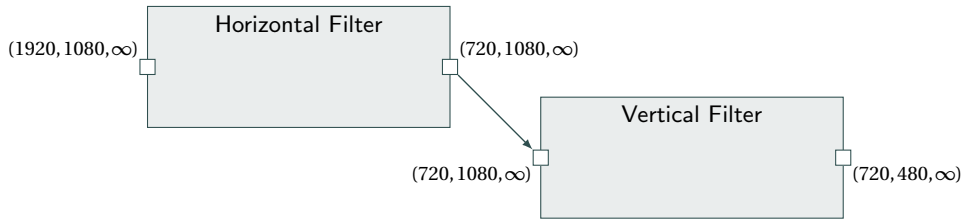
Formally, an Array-OL application is a set of *tasks* connected through *ports*. The tasks are equivalent to mathematical functions reading data on their input ports and writing data on their output ports. The tasks are of three kinds: *elementary*, *compound* and *repetition*. An elementary task is atomic (a black box), it can come from a library for example. A compound is a dependence graph whose nodes are tasks connected via their ports. A repetition is a task expressing how a single sub-task is repeated.

All the data exchanged between the tasks are arrays. These arrays are multidimensional and are characterized by their *shape*, the number of elements on each of their dimension[1]. A shape will be noted as a column vector or a comma-separated tuple of values indifferently. Each port is thus characterized by the shape and the type of the elements of the array it reads from or writes to. As said above, the Array-OL model is single assignment. One manipulates *values* and not *variables*. Time is thus represented as one (or several) dimension of the data arrays. For example, an array representing a video is three-dimensional of shape (width of frame, height of frame, frame number). We will illustrate the rest of the presentation of Array-OL by an application that scales an high definition TV signal down to a standard definition TV signal. Both signals will be represented as a three dimensional array.

2.2 Task parallelism

The task parallelism is represented by a compound task. The compound description is a simple *directed acyclic graph*. Each node represents a task and each edge a dependence connecting two conform ports (same type and shape). There is no relation between the shapes of the inputs and the outputs of a task. So a task can read two two-dimensional arrays and write a three-dimensional one. The creation of dimensions by a task is very useful, a very simple example is the FFT which creates a frequency dimension. We will study as a running example a downscaler from high definition TV to standard definition TV. Here is the top level compound description. The tasks are represented by named rectangles, their ports are squares on the border of the tasks. The shape of the ports is written as a t-uple of positive numbers or ∞. The dependences are represented by arrows between ports.

[1] A point, seen as a 0-dimensional array is of shape $()$, seen as a 1-dimensional array is of shape (1), seen as a 2-dimensional array is of shape $\binom{1}{1}$, etc.



There is only one limitation on the dimensions: there must be at most one infinite dimension by array. Most of the time, this infinite dimension is used to represent the time, so having only one is quite sufficient.

Each execution of a task reads one full array on its inputs and writes the full output arrays. It's not possible to read more than one array per port to write one. *The graph is a dependence graph, not a data flow graph.*

So it is possible to schedule the execution of the tasks just with the compound description. But it's not possible to express the data parallelism of our applications because the details of the computation realized by a task are hidden at this specification level.

2.3 Data parallelism

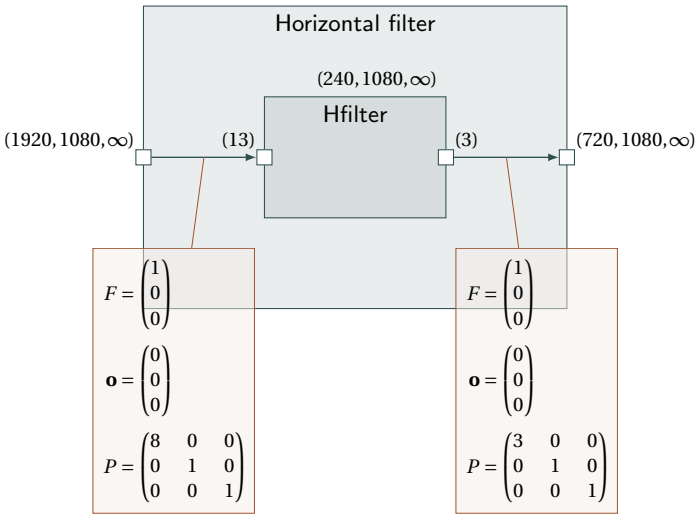
A data-parallel repetition of a task is specified in a repetition task. The basic hypothesis is that all the repetitions of this repeated task are independent. They can be scheduled in any order, even in parallel [2]. The second one is that each instance of the repeated task operates with sub-arrays of the inputs and outputs of the repetition. For a given input or output, all the sub-array instances have the same shape, are composed of regularly spaced elements and are regularly placed in the array. This hypothesis allows a compact representation of the repetition and is coherent with the application domain of Array-OL which describes very regular algorithms.

As these sub-arrays are conform, they are called *patterns* when considered as the input arrays of the repeated task and *tiles* when considered as a set of elements of the arrays of the repetition task. In order to give all the information needed to create these patterns, a *tiler* is associated to each array (ie each edge). A tiler is able to build the patterns from an input array, or to store the patterns in an output array. It describes the coordinates of the elements of the tiles from the coordinates of the elements of the patterns. It contains the following information:

- F: a *fitting* matrix.
- $\mathbf{o}$: the *origin* of the *reference pattern* (for the *reference repetition*).
- P: a *paving* matrix.

[2] This is why we talk of *repetitions* and not *iterations* which convey a sequential semantics.

Visual representation of a repetition task. The shapes of the arrays and patterns are, as in the compound description, noted on the ports. The *repetition space* indicating the number of repetitions is defined itself as an multidimensional array with a shape. Each dimension of this repetition space can be seen as a parallel loop and the shape of the repetition space gives the bounds of the loop indices of the nested parallel loops. An example of the visual description of a repetition is given below by the horizontal filter repetition from the downscaler. The tilers are connected to the dependences linking the arrays to the patterns. Their meaning is explained below.

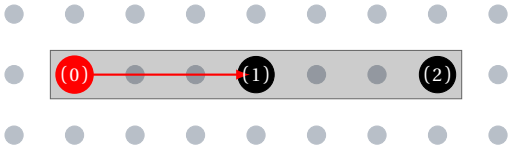


Building a tile from a pattern. From a *reference element* (**ref**) in the array, one can extract a pattern by enumerating its other elements relatively to this reference element. The *fitting* matrix is used to compute the other elements. The coordinates of the elements of the pattern ($\mathbf{e_i}$) are built as the sum of the coordinates of the reference element and a linear combination of the fitting vectors as follows

$$\forall \mathbf{i}, \mathbf{0} \leq \mathbf{i} < \mathbf{s}_{\text{pattern}}, \mathbf{e_i} = \mathbf{ref} + F \cdot \mathbf{i} \mod \mathbf{s}_{\text{array}} \quad (1)$$

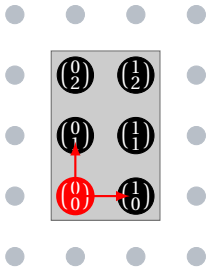
where $\mathbf{s}_{\text{pattern}}$ is the shape of the pattern, $\mathbf{s}_{\text{array}}$ is the shape of the array and F the fitting matrix.

In the following examples of fitting matrices and tiles, the tiles are drawn from a reference element in a 2D array. The array elements are labeled by their index in the pattern, **i**, illustrating the formula $\forall \mathbf{i}, \mathbf{0} \leq \mathbf{i} < \mathbf{s}_{\text{pattern}}, \mathbf{e_i} = \mathbf{ref} + F \cdot \mathbf{i}$. The fitting vectors constituting the basis of the tile are drawn from the reference point.



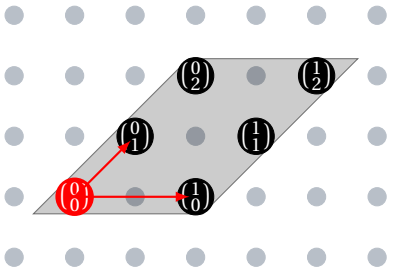
$F = \begin{pmatrix} 3 \\ 0 \end{pmatrix}$ $\mathbf{s}_{\text{pattern}} = (3)$

There are here 3 elements in this tile because the shape of the pattern is (3). The indices of these elements are thus (0), (1) and (2). Their position in the tile relatively to the reference point are thus $F \cdot (0) = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, F \cdot (1) = \begin{pmatrix} 3 \\ 0 \end{pmatrix}, F \cdot (2) = \begin{pmatrix} 6 \\ 0 \end{pmatrix}$.



$F = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ $\mathbf{s}_{\text{pattern}} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$

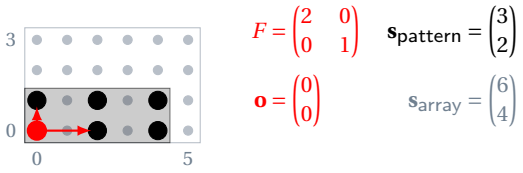
The pattern is here two-dimensional with 6 elements. The fitting matrix builds a compact rectangular tile in the array.



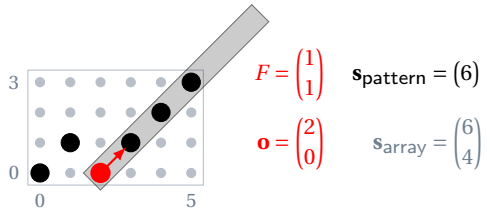
$F = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix}$ $\mathbf{s}_{\text{pattern}} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$

This last example illustrates how the tile can be sparse, thanks to the $\begin{pmatrix} 2 \\ 0 \end{pmatrix}$ fitting vector, and non parallel to the axes of the array, thanks to the $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ fitting vector.

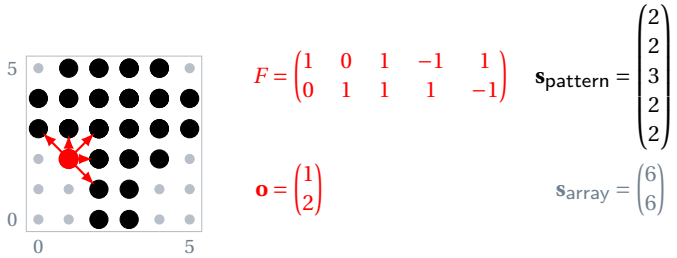
A key element one has to remember when using Array-OL is that all the dimensions of the arrays are toroidal. That means that all the coordinates of the tile elements are computed modulo the size of the array dimensions. The following more complex examples of tiles are drawn from a fixed reference element ($\mathbf{o}$ as origin in the figure) in fixed size arrays, illustrating the formula $\forall \mathbf{i}, \mathbf{0} \leq \mathbf{i} < \mathbf{s}_{\text{pattern}}, \mathbf{e_i} = \mathbf{o} + F \cdot \mathbf{i} \mod \mathbf{s}_{\text{array}}$.



A sparse tile aligned on the axes of the array.



The pattern is here mono-dimensional, the fitting builds a diagonal tile that wraps around the array because of the modulo.



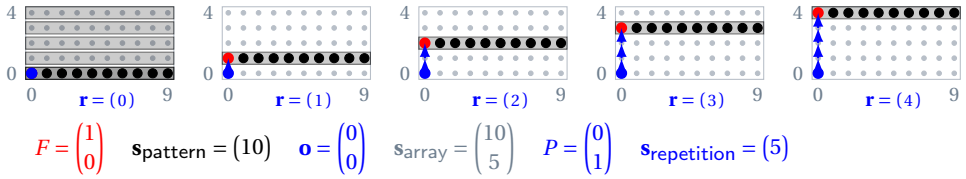
This is an extreme case of a five-dimensional pattern fitted as a two-dimensional tile. Most of the elements of the tile are read several times to build the 48 pattern elements.

Paving an array with tiles. For each repetition, one needs to design the reference elements of the input and output patterns. A similar scheme as the one used to enumerate the elements of a pattern is used for that purpose.

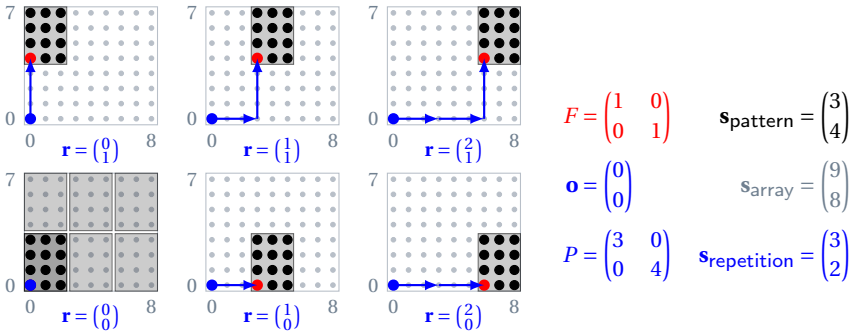
The reference elements of the reference repetition are given by the *origin* vector, $\mathbf{o}$, of each tiler. The reference elements of the other repetitions are built relatively to this one. As above, their coordinates are built as a linear combination of the vectors of the *paving* matrix as follows

$$\forall \mathbf{r}, \mathbf{0} \leq \mathbf{r} < \mathbf{s}_{\text{repetition}}, \mathbf{ref}_{\mathbf{r}} = \mathbf{o} + P \cdot \mathbf{r} \mod \mathbf{s}_{\text{array}} \tag{2}$$

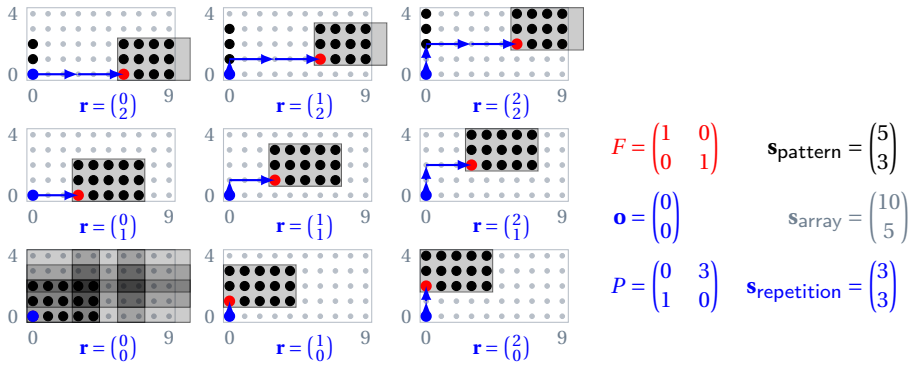
where $\mathbf{s}_{\text{repetition}}$ is the shape of the repetition space, P the paving matrix and $\mathbf{s}_{\text{array}}$ the shape of the array. Here are some examples.



This figure represents the tiles for all the repetitions in the repetition space, indexed by $\mathbf{r}$. The paving vectors drawn from the origin $\mathbf{o}$ indicate how the coordinates of the reference element $\mathbf{ref_r}$ of the current tile are computed. Here the array is tiled row by row.



A 2D pattern tiling exactly a 2D array.



The tiles can overlap and the array is toroidal.

Summary. We can summarize all these explanations with two formulas:

- $\forall \mathbf{r}, \mathbf{0} \leq \mathbf{r} < \mathbf{s}_{\text{repetition}}, \mathbf{ref_r} = \mathbf{o} + P \cdot \mathbf{r} \mod \mathbf{s}_{\text{array}}$ gives all the reference elements of the patterns,
- $\forall \mathbf{i}, \mathbf{0} \leq \mathbf{i} < \mathbf{s}_{\text{pattern}}, \mathbf{e_i} = \mathbf{ref_r} + F \cdot \mathbf{i} \mod \mathbf{s}_{\text{array}}$ enumerates all the elements of a pattern for repetition $\mathbf{r}$,

where $\mathbf{s}_{\text{array}}$ is the shape of the array, $\mathbf{s}_{\text{pattern}}$ is the shape of the pattern, $\mathbf{s}_{\text{repetition}}$ is the shape of the repetition space, $\mathbf{o}$ is the coordinates of the reference element of the reference pattern, also called the origin, P is the paving matrix whose column vectors, called the paving vectors, represent the regular spacing between the patterns, F is the fitting matrix whose column vectors, called the fitting vectors, represent the regular spacing between the elements of a pattern in the array.

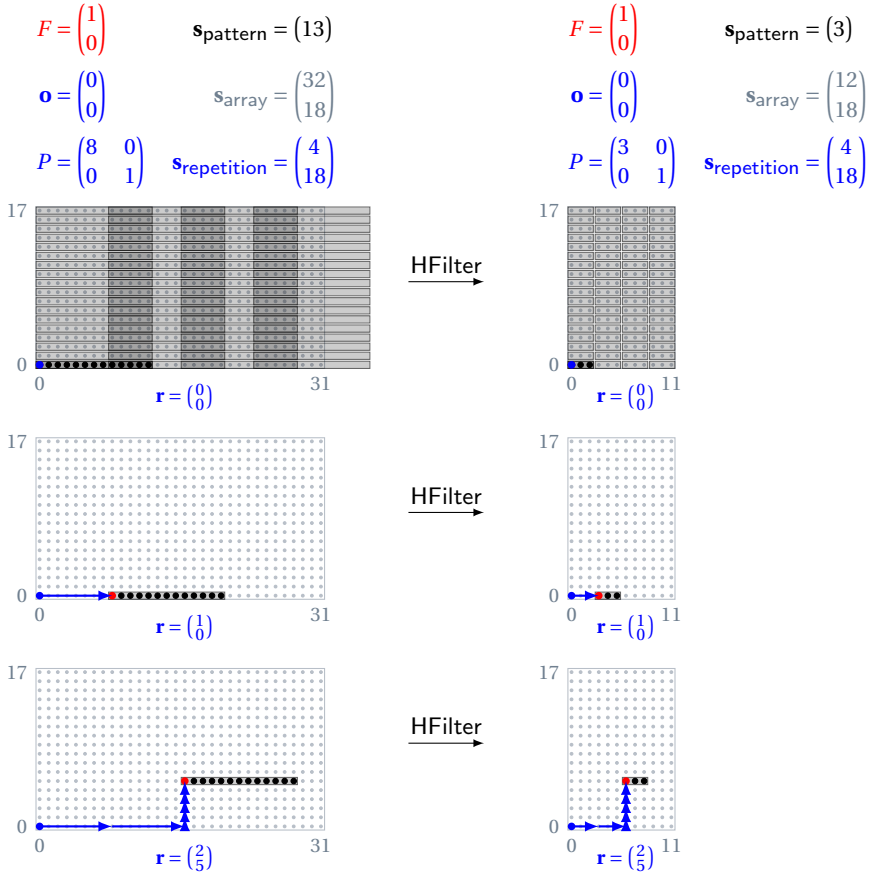
Some constraints on the number of rows and columns of the matrices can be derived from their use. The origin, the fitting matrix and the paving matrix have a number of rows equal to the dimension of the array; the fitting matrix has a number of columns equal to the dimension of the pattern [3]; and the paving matrix has a number of columns equal to the dimension of the repetition space.

Linking the inputs to the outputs by the repetition space. The previous formulas explain which element of an input or output array one repetition consumes or produces. The link between the inputs and outputs is made by the repetition index, $\mathbf{r}$. For a given repetition, the output patterns (of index $\mathbf{r}$) are produced by the repeated task from the input patterns (of index $\mathbf{r}$). These pattern elements correspond to array elements through the tiles associated to the patterns. Thus the set of tilers and the shapes of the patterns and repetition space define the dependences between the elements of the output arrays and the elements of the input arrays of a repetition. As stated before, no execution order is implied by these dependences between the repetitions.

To illustrate this link between the inputs and the outputs, we show below several repetitions of the horizontal filter repetition. In order to simplify the figure and as the treatment is made frame by frame, only the first two dimensions are represented [4]. The sizes of the arrays have also been reduced by a factor of 60 in each dimension for readability reasons.

[3] Thus if the pattern is a single element viewed as a zero-dimensional array, the fitting matrix is empty and noted as $()$. The only element of a tile is then its reference element. This can be viewed as a degenerate case of the general fitting equation where there is no index $\mathbf{i}$ and so no multiplication $F \cdot \mathbf{i}$.

[4] Indeed, the third dimension of the input and output arrays is infinite, the third dimension of the repetition space is also infinite, the tiles do not cross this dimension and the only paving vector having a non null third element is $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ along the infinite repetition space dimension.



2.4 Enforcing determinism by construction

The basic design decision that enforces determinism is the fact that Array-OL only expresses data dependences. To ease the manipulation of the values, the language is single assignment. Thus each array element has to be written only once. To simplify the verification of this, the constraint that each task produces all the elements of its output arrays is built into the model. An array has to be fully produced even if some elements are not read by any other task. Enforcing this rule for all the tasks at all the levels of the hierarchy also allows to compose tasks easily. A direct consequence of this full production rule is that a repetition has to tile exactly its output arrays. In other words each element of an output array has to belong to exactly one tile. Verifying this can be done by using polyhedra computations using a tool like SPPoC[5] [3].

[5] `http://www.lifl.fr/west/sppoc/`

To check that all the elements of an output array have been produced, one can check that the union of the tiles spans the array. The union of all the tiles can be built as the set of points $\mathbf{e}_{(\mathbf{r},\mathbf{i})}$ verifying the following system of (in)equations

$$\begin{cases} \mathbf{0} \leq \mathbf{r} < \mathbf{s}_{\text{repetition}} \\ \mathbf{ref}_{\mathbf{r}} = \mathbf{o} + P \cdot \mathbf{r} \mod \mathbf{s}_{\text{array}} \\ \mathbf{0} \leq \mathbf{i} < \mathbf{s}_{\text{pattern}} \\ \mathbf{e}_{(\mathbf{r},\mathbf{i})} = \mathbf{ref}_{\mathbf{r}} + F \cdot \mathbf{i} \mod \mathbf{s}_{\text{array}} \end{cases} . \tag{3}$$

Building the difference between the array and this set is done in one operation (polyhedral difference from the Polylib[6] that is included in SPPoC) and testing if the resulting set is empty is done by looking for an element in this set using a call to the PIP[7] [12] solver that is also included in SPPoC. These operations are possible because, as the shapes are known values, the system of inequations is equivalent to a system of affine equations.

To check that no point is computed several times in an output array, one builds the following set of points, e, (intersection of two tiles) verifying the following system of (in)equations

$$\begin{cases} \mathbf{0} \leq \mathbf{r} < \mathbf{s}_{\text{repetition}} \\ \mathbf{ref}_{\mathbf{r}} = \mathbf{o} + P \cdot \mathbf{r} \mod \mathbf{s}_{\text{array}} \\ \mathbf{0} \leq \mathbf{i} < \mathbf{s}_{\text{pattern}} \\ \mathbf{e} = \mathbf{ref}_{\mathbf{r}} + F \cdot \mathbf{i} \mod \mathbf{s}_{\text{array}} \\ \mathbf{0} \leq \mathbf{r}' < \mathbf{s}_{\text{repetition}} \\ \mathbf{ref}_{\mathbf{r}'} = \mathbf{o} + P \cdot \mathbf{r}' \mod \mathbf{s}_{\text{array}} \\ \mathbf{0} \leq \mathbf{i}' < \mathbf{s}_{\text{pattern}} \\ \mathbf{e} = \mathbf{ref}_{\mathbf{r}'} + F \cdot \mathbf{i}' \mod \mathbf{s}_{\text{array}} \end{cases} . \tag{4}$$

If this set is empty, then no two tiles overlap and each computed element is computed once. To check the emptiness of this set, the same technique as above can be used: to call PIP. As above, the above system of inequations is equivalent to a system of affine equations, thus solvable by PIP.

With these two checks, one can ensure that all the elements of the output arrays are computed exactly once and so that the single assignment is respected.

We have defined in this section the Array-OL language, its principles and how it allows to express in a deterministic way task and data parallelism. The most original feature of Array-OL is the description of the array accesses in data parallel repetitions by tiling. As this language make no assumption on the execution platform, we will study in the next section how the projection of an Array-OL specification to such an execution platform can be made.

3 Projection onto an execution model

The Array-OL language expresses the minimal order of execution that leads to the correct computation. This is a design intension and lots of decisions can and have to be taken when mapping an

[6] `http://icps.u-strasbg.fr/polylib/`
[7] `http://www.piplib.org/`

Array-OL specification onto an execution platform: how to map the various repetition dimensions to time and space, how to place the arrays in memory, how to schedule parallel tasks on the same processing element, how to schedule the communications between the processing elements?

3.1 Space-time mapping

One of the basic questions one has to answer is: What dimensions of a repetition should be mapped to different processors or to a sequence of steps? To be able to answer this question, one has to look at the environment with which the Array-OL specification interacts. If a dimension of an array is produced sequentially, it has to be projected to time, at least partially. Some of the inputs could be buffered and treated in parallel. On the contrary, if a dimension is produced in parallel (e.g. by different sensors), it is natural to map it to different processors. But one can also group some repetitions on a smaller number of processors and execute these groups sequentially. The decision is thus also influenced by the available hardware platform.

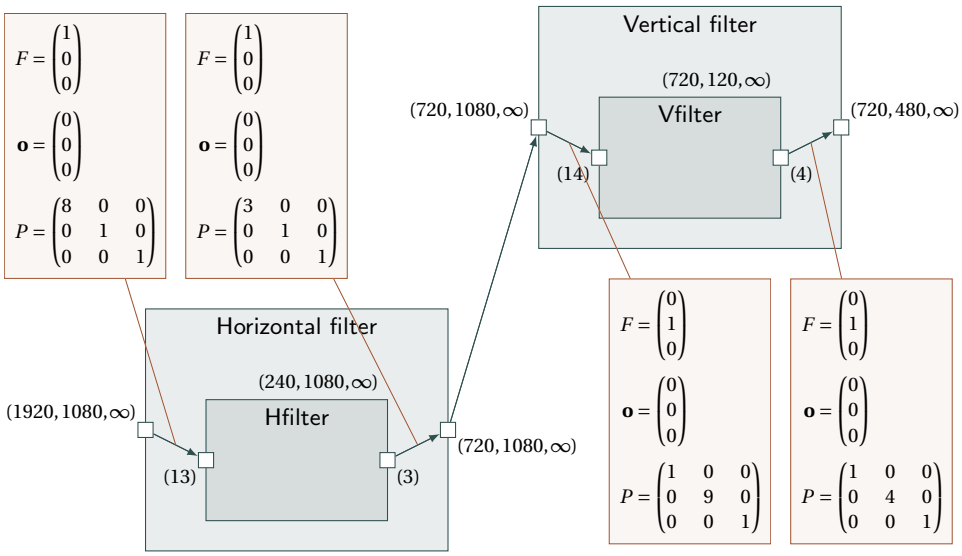
It is a strength of Array-OL that the space-time mapping decision is separated from the functional specification. This allows to build functional component libraries for reuse and to carry out some architecture exploration with the least restrictions possible.

Mapping compounds is not specially difficult. The problem comes when mapping repetitions. This problem is discussed in details in [1] where the authors study the projection of Array-OL onto Kahn process networks [14, 15]. The key point is that some repetitions can be transformed to flows. In that case, the execution of the repetitions is sequentialized (or pipelined) and the patterns are read and written as a flow of tokens (each token carrying a pattern).

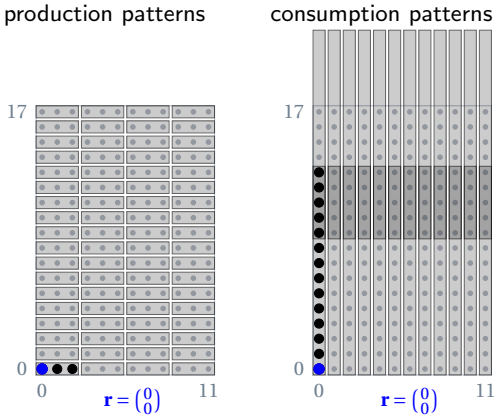
3.2 Transformations

A set of Array-OL code transformations has been designed to allow to adapt the application to the execution, allowing to choose the granularity of the flows and a simple expression of the mapping by tagging each repetition by its execution mode: data-parallel or sequential.

These transformations allow to cope with a common difficulty of multidimensional signal processing applications: how to chain two repetitions, one producing an array with some paving and the other reading this same array with another paving? To better understand the problem, let us come back to the downscaler example where the horizontal filter produces a $(720, 1080, \infty)$ array row-wise 3 by 3 elements and the vertical filter reads it column-wise 14 elements by 14 elements with a sliding overlap between the repetitions as shown on the following figure.



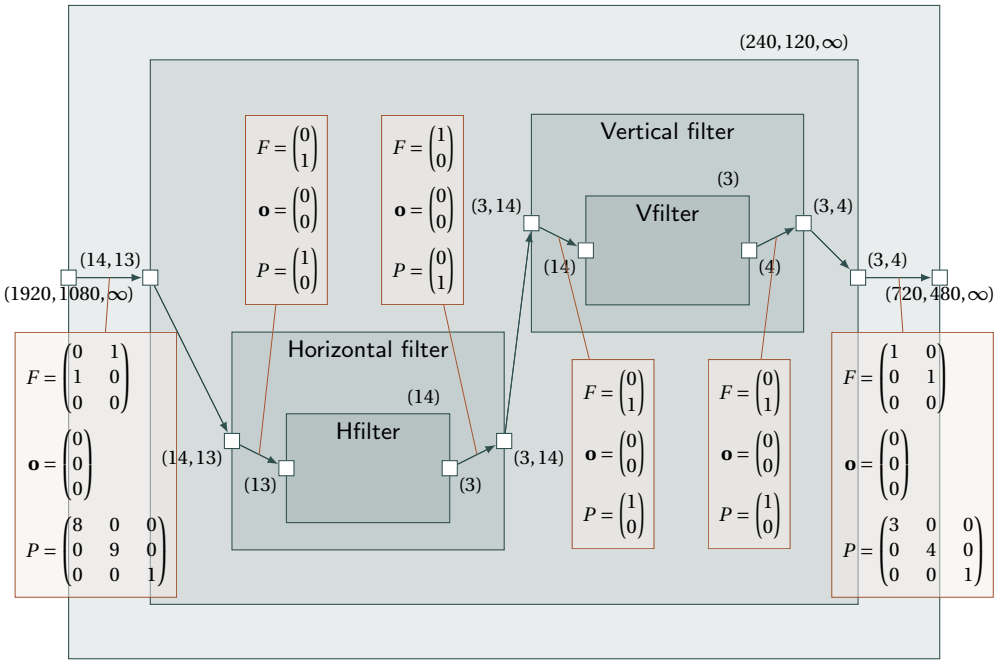
The interesting array is the intermediate $(720, 1080, \infty)$ array that is produced by tiles of 3 elements aligned along the first dimension and consumed by tiles of 13 elements aligned on the second dimension.



$\frac{1}{60}$-th of the first two-dimensions and suppression of the infinite dimension of the intermediate $(720, 1080, \infty)$ array.

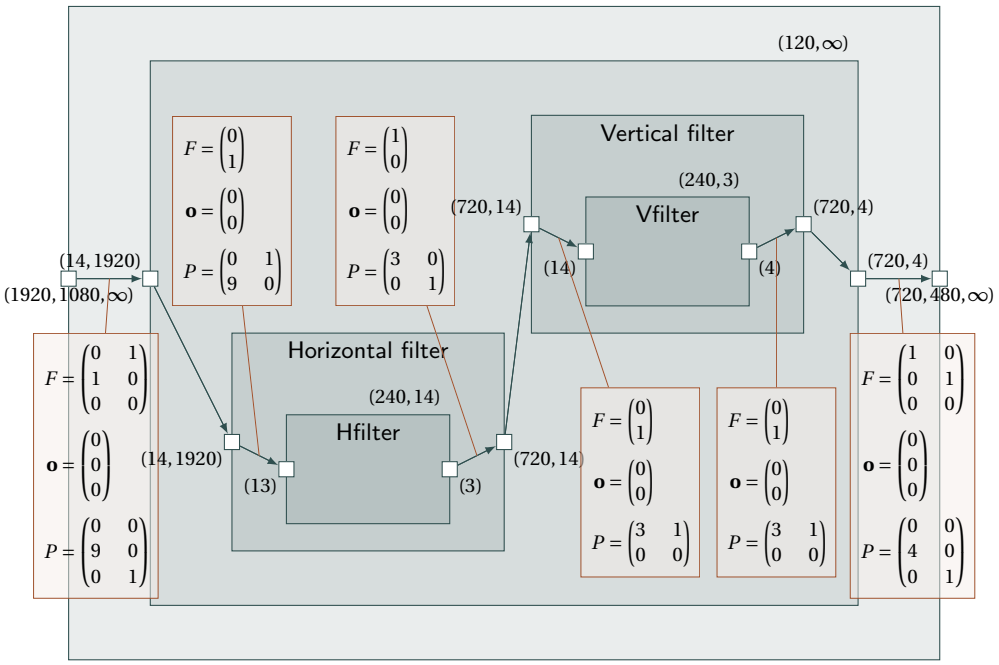
In order to be able to project this application onto an execution platform, one possibility is to make a flow of the time dimension and to allow pipelining of the space repetitions. A way to do

that is to transform the application by using the fusion transformation to add a hierarchical level. The top level can then be transformed into a flow and the sub-level can be pipelined. Here is the transformed application.



A hierarchical level has been created that is repeated $(240,120,\infty)$ times. The intermediate array between the filters has been reduced to the minimal size that respects the dependences. If the inserted level is executed sequentially and if the two filters are executed on different processors, the execution can be pipelined.

This form of the application takes into account internal constraints: how to chain the computations. Now, the environment tells us that a TV signal is a flow of pixels, row after row. We can now propose a new form of the downscaler application taking that environment constraint into account by extending the top-level patterns to include full rows. Here is what such an application could look like.



The top-level repetition now works with tiles containing full rows of the images. Less parallelism is expressed at that level but as the images arrive in the system row by row, the buffering mechanism is simplified and the full parallelism is still available at the lower level.

A full set of transformations (fusion, tiling, change paving, collapse) described in [8] allows to adapt the application to the execution platform in order to build an efficient schedule compatible with the internal computation chaining constraints, those of the environment and the possibilities of the hardware. A great care has been taken in these transformations to ensure that they do not modify the semantics of the specifications. They only change the way the dependences are expressed in different hierarchical levels but not the precise element to element dependences.

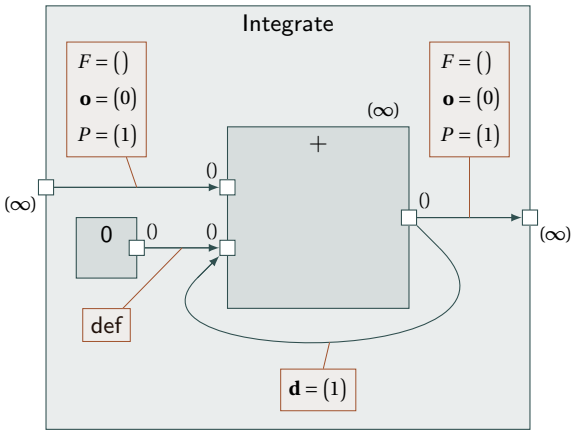
4 Extensions

Around the core Array-OL language, several extensions have been proposed recently. We will give here the basic ideas of these extensions and pointers to references where the reader can go into details.

4.1 Inter-Repetition dependences

To be able to represent loops containing inter-repetition dependencies, we have added the possibility to model uniform dependencies between tiles produced by the repeated component and

tiles consumed by this repeated component. The simplest example is the discrete integration shown below.



Here the patterns (and so the tiles) are single points. The uniform dependence vector $\mathbf{d} = (1)$ tells that repetition $\mathbf{r}$ depends on repetition $\mathbf{r} - \mathbf{d}(= \mathbf{r} - (1))$ by adding the result of the addition of index $\mathbf{r} - (1)$ to the input tile $\mathbf{r}$. This is possible because the output pattern and input pattern linked by the inter-repetition dependence connector have the same shape. To start the computation, a default value of 0 is taken for repetition 0.

Formally an inter-repetition dependence connects an output port of a repeated component with one of its input ports. The shape of these connected ports must be identical. The connector is tagged with a dependence vector $\mathbf{d}$ that defines the dependence distance between the dependent repetitions. This dependence is uniform, that means identical for all the repetitions. When the source of a dependence is outside the repetition space, a default value is used. This default value is defined by a connector tagged with "def".

4.2 Control modeling

In order to model mixed control flow, data flow applications, Labbani et al. [17, 18]have proposed to use the mode automata concept. An adaptation of this concept to Array-OL is necessary to couple an automaton and modes described as Array-OL components corresponding to the states of that automaton.

A *controlled component* is a switch allowing to select one component according to a special "mode" input. All the selectable components must have the same interface (same number and types of ports). An *automaton component* produces a 1D array of values that will be used as mode inputs of a controlled component. A repetition component allows to associate the mode values to a repetition of a controlled component.

Both the inter-repetition and the control modeling extensions can be used at any level of hierarchy, thus allowing to model complex applications. The Array-OL transformations still need to be extended to deal with these extensions.

5 Tools

Several tools have been developed using the Array-OL language as specification language. Gaspard Classic[8] [5] takes as input an Array-OL specification, allows the user to apply transformations to it, and generates multi-threaded C++ code allowing to execute the specification on a shared memory multi-processor computer.

The Gaspard2[9] co-modeling environment [2] aims at proposing a model-driven environment to co-design intensive computing systems-on-chip. It proposes a UML profile to model the application, the hardware architecture and the allocation of the application onto the architecture. The application metamodel is based on Array-OL with the inter-repetition dependence and control modeling extensions. The hardware metamodel takes advantage of the repetition mechanism proposed by Array-OL to model repetitive hardware components such as SIMD units, multi-bank memories or networks-on-chip. The allocation mechanism also builds upon the Array-OL constructs to express data-parallel distributions. The Gaspard2 tool is built as an Eclipse[10] plugin and mainly generates SystemC code for the co-simulation of the modeled system-on-chip. It also includes an improved transformation engine.

Two smaller tools are also available[11]: a simulation [10] of Array-OL in PtolemyII [22] and Array-OL example, a pedagogical tool helping to visualize repetitions in 3D. And to be complete, we have to mention that Thales has developed its own internal tools using Array-OL to develop radar and sonar applications on multiprocessor platforms.

Acknowledgment

The author would like to thank all the members of the west team of the laboratoire d'informatique fondamentale de Lille who have worked on the definition and compilation of Array-OL or used it as a tool for their work. They have also made some very useful comments on drafts of this paper.

6 Conclusion

We have presented in this paper the Array-OL language. This language is dedicated to specify intensive signal processing applications. It allows to model the full parallelism of the application: both task and data parallelisms. Array-OL is a single assignment first order functional language manipulating multidimensional arrays. It focuses on the expression of the main difficulty of

[8] `http://www2.lifl.fr/west/gaspard/classic.html`
[9] `http://www2.lifl.fr/west/gaspard/`
[10] `http://www.eclipse.org/`
[11] `http://www2.lifl.fr/west/aoltools/`

the intensive signal processing applications: the multidimensional data accesses. It proposes a mechanism able to express at a high level of abstraction the regular tilings of the arrays by data-parallel repetitions. The original Array-OL language has been extended to support inter-repetition dependences and some control modeling.

As an Array-OL specification describes the minimal order of computing, its space-time mapping has to be done taking into account constraints that are not expressed in Array-OL: architectural and environmental constraints. A toolbox of code transformations allows to adapt the application to its deployment environment. Future works include extending this toolbox to handle the control extensions and automating the allocation process of an application on a distributed heterogeneous platform in the Gaspard2 co-modeling environment.

References

[1] Abdelkader Amar, Pierre Boulet, and Philippe Dumont. Projection of the Array-OL specification language onto the Kahn process network computation model. In *International Symposium on Parallel Architectures, Algorithms, and Networks*, Las Vegas, Nevada, USA, December 2005.

[2] Pierre Boulet, Cédric Dumoulin, and Antoine Honoré. *From MDD concepts to experiments and illustrations*, chapter Model Driven Engineering for System-on-Chip Design. ISTE, International scientificc and technical encyclopedia, Hermes science and Lavoisier, September 2006.

[3] Pierre Boulet and Xavier Redon. SPPoC : manipulation automatique de polyèdres pour la compilation. *Technique et Science Informatiques*, 20(8):1019–1048, 2001. (In French).

[4] Michael J. Chen and Edward A; Lee. Design and implementation of a multidimensional synchronous dataflow environment. In *1995 Proc. IEEE Asilomar Conf. on Signal, Systems, and Computers*, 1995.

[5] Jean-Luc Dekeyser, Philippe Marquet, and Julien Soula. Video kills the radio stars. In *Supercomputing'99 (poster session)*, Portland, OR, November 1999. (`http://www.lifl.fr/west/gaspard/`).

[6] Alain Demeure and Yannick Del Gallo. An Array Approach for Signal Processing Design. In *Sophia-Antipolis conference on Micro-Electronics (SAME 98)*, France, October 1998.

[7] Alain Demeure, Anne Lafarge, Emmanuel Boutillon, Didier Rozzonelli, Jean-Claude Dufourd, and Jean-Louis Marro. Array-OL : Proposition d'un formalisme tableau pourle traitement de signal multi-dimensionnel. In *Gretsi*, Juan-Les-Pins, France, September 1995.

[8] Philippe Dumont. *Spécification Multidimensionnelle pour le traitement du signal systématique*. Thèse de doctorat (PhD Thesis), Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, December 2005.

[9] Philippe Dumont and Pierre Boulet. Another multidimensional synchronous dataflow: Simulating Array-OL in ptolemy II. Research Report RR-5516, INRIA, March 2005. `http://www.inria.fr/rrrt/rr-5516.html`.

[10] Philippe Dumont and Pierre Boulet. Another multidimensional synchronous dataflow, simulating Array-OL in PtolemyII. to appear, 2005.

[11] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proc. of the IEEE*, 85(3), year 1997.

[12] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[13] Axel Jantsch and Ingo Sander. Models of computation and languages for embedded system design. *IEE Proceedings on Computers and Digital Techniques*, 152(2):114–129, March 2005. Special issue on Embedded Microelectronic Systems; Invited paper.

[14] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland, August 1974.

[15] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77: Proceedings of the IFIP Congress 77*, pages 993–998. North-Holland, 1977.

[16] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, July 1967.

[17] Ouassila Labbani, Jean-Luc Dekeyser, Pierre Boulet, and Éric Rutten. UML2 profile for modeling controlled data parallel applications. In *FDL'06: Forum on specification and Design Languages*, Darmstadt, Germany, September 2006.

[18] Ouassila Labbani, Jean-Luc Dekeyser, Pierre Boulet, and Éric Rutten. Introducing control in the gaspard2 data-parallel metamodel: Synchronous approach. *International Workshop MARTES: Modeling and Analysis of Real-Time and Embedded Systems (in conjunction with 8th International Conference on Model Driven Engineering Languages and Systems, MoDELS/UML 2005)*, October 2005.

[19] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, January 1987.

[20] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. In *Proc. of the IEEE*, September 1987.

[21] Edward A. Lee. Multidimensional streams rooted in dataflow. In *Proceedings of the IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Orlando, Florida, January 1993. North-Holland.

[22] Edward A. Lee. *Overview of the Ptolemy Project.* University of California, Berkeley, March 2001.

[23] Christophe Mauras. *Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, Université de Rennes I, December 1989.

[24] Praveen K. Murthy and Edward A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50(8):2064–2079, August 2002.

[25] Praveen Kumar Murthy. *Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow.* PhD thesis, University of California, Berkeley, CA, 1996.

[26] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, 1998.

[27] Julien Soula. *Principe de Compilation d'un Langage de Traitement de Signal.* Thèse de doctorat (PhD Thesis), Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, December 2001. (In French).

[28] Julien Soula, Philippe Marquet, Jean-Luc Dekeyser, and Alain Demeure. Compilation principle of a specification language dedicated to signal processing. In *Sixth International Conference on Parallel Computing Technologies, PaCT 2001*, pages 358–370, Novosibirsk, Russia, September 2001. Lecture Notes in Computer Science vol. 2127.

Contents

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399